Dart Programming - Functions

Functions are the building blocks of readable, maintainable, and reusable code. A function is a set of statements to perform a specific task. Functions organize the program into logical blocks of code. Once defined, functions may be called to access code. This makes the code reusable. Moreover, functions make it easy to read and maintain the program's code.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Sr.No	Functions & Description
1	Defining a Function A function definition specifies what and how a specific task would be done.
2	Calling a Function A function must be called so as to execute it.
3	Returning Functions Functions may also return value along with control, back to the caller.
4	Parameterized Function Parameters are a mechanism to pass values to functions.

Optional Parameters

Optional parameters can be used when arguments need not be compulsorily passed for a function's execution. A parameter can be marked optional by appending a question mark to its name. The optional parameter should be set as the last argument in a function.

We have three types of optional parameters in Dart −

Sr.No	Parameter & Description
1	Optional Positional Parameter To specify optional positional parameters, use square [] brackets.
2	Optional named parameter Unlike positional parameters, the parameter's name must be specified while the value is being passed. Curly brace {} can be used to specify optional named parameters.
3	Optional Parameters with Default Values Function parameters can also be assigned values by default. However, such parameters can also be explicitly passed values.

Recursive Dart Functions

Recursion is a technique for iterating over an operation by having a function call to itself repeatedly until it arrives at a result. Recursion is best applied when you need to call the same function repeatedly with different parameters from within a loop.

Example

```
void main() {
   print(factorial(6));
}
factorial(number) {
   if (number <= 0) {
      // termination case
      return 1;
   } else {
      return (number * factorial(number - 1));
      // function invokes itself
   }
}
```

It should produce the following **output** −

```
720
```

Lambda Functions

Lambda functions are a concise mechanism to represent functions. These functions are also called as Arrow functions.

Syntax

```
[return_type]function_name(parameters)=>expression;
```

Example

```
void main() {
   printMsg();
   print(test());
}
printMsg()=>
print("hello");

int test()=>123;
// returning function
```

It should produce the following **output** −

```
hello 123
```

Dart Programming - Defining a Function

A function definition specifies what and how a specific task would be done. Before using a function, it must be defined. The syntax for defining a standard function is given below −

```
function_name() {
   //statements
}
```

OR

```
void function_name() {
   //statements
}
```

The **void** keyword indicates that the function does not return any value to the caller.

Example - Simple Function Definition

```
test() {
   //function definition
   print("function called"); }
```

Dart Programming - Calling a Function

A function must be called to execute it. This process is termed as **function invocation**.

Syntax

```
function_name()
```

The following example illustrates how a function can be invoked −

```
void main() {
   test();
}
test() {
   //function definition
   print("function called");
}
```

It will produce the following **output** −

```
function called
```

Dart Programming - Returning Function

Functions may also return value along with the control, back to the caller. Such functions are called as **returning functions**.

Syntax

```
return_type function_name(){
   //statements
   return value;
}
```

- The **return_type** can be any valid data type.
- The **return** statement is optional. I not specified the function returns null;
- The data type of the value returned must match the return type of the function.
- A function can return at the most one value. In other words, there can be only one return statement per function.

Example

Let's take an example to understand how **returning functions** work.

- The example declares a function **test()**. The function's return type is string.
- The function returns a string value to the caller. This is achieved by the return statement.
- The function **test()** returns a string. This is displayed as output.

```
void main() {
   print(test());
}
String test() {
   // function definition
   return "hello world";
}
```

It will produce the following **output** −

```
hello world
```

Dart Programming - Parameterized Function

Parameters are a mechanism to pass values to functions. Parameters form a part of the function's signature. The parameter values are passed to the function during its invocation. Unless explicitly specified, the number of values passed to a function must match the number of parameters defined.

Let us now discuss the ways in which parameters can be used by functions.

Required Positional Parameters

It is mandatory to pass values to required parameters during the function call.

Syntax

```
Function_name(data_type param_1, data_type param_2[…]) {
   //statements
}
```

Example

The following code snippet declares a function **test_param** with two parameters namely, **n1** and **s1**

- It is not mandatory to specify the data type of the parameter. In the absence of a data type, the parameters type is determined dynamically at runtime.
- The data type of the value passed must match the type of the parameter during its declaration. In case the data types don't match, the compiler throws an error.

```
void main() {
   test_param(123,"this is a string");
}
test_param(int n1,String s1) {
   print(n1);
   print(s1);
}
```

The **output** of the above code is as follows −

```
123
this is a string
```

Optional Positional Parameter

To specify optional positional parameters, use square [] brackets.

Syntax

```
void function_name(param1, [optional_param_1, optional_param_2]) { }
```

If an optional parameter is not passed a value, it is set to NULL.

Example

```
void main() {
   test_param(123);
}
test_param(n1,[s1]) {
   print(n1);
   print(s1);
}
```

It should produce the following **output–**

```
123
null
```

Optional Named Parameter

Unlike positional parameters, the parameters' name must be specified while the value is being passed. Curly brace {} can be used to specify optional named parameters.

Syntax - Declaring the function

```
void function_name(a, {optional_param1, optional_param2}) { }
```

Syntax - Calling the function

```
function_name(optional_param:value,…);
```

Example

```
void main() {
   test_param(123);
   test_param(123,s1:'hello');
   test_param(123,s2:'hello',s1:'world');
}
test_param(n1,{s1,s2}) {
   print(n1);
   print(s1);
}
```

It should produce the following **output**−

```
123
null
123
hello
123
world
```

Optional Parameters with Default Values

Function parameters can also be assigned values by default. However, such parameters can also be explicitly passed values.

Syntax

```
function_name(param1,{param2= default_value}) {
   //......
}
```

Example

```
void main() {
   test_param(123);
}
void test_param(n1,{s1:12}) {
   print(n1);
   print(s1);
}
```

It should return the following **output**−

```
123
12
```

Note − All required parameters in a function must occur before optional parameters.